Available online at www.sciencedirect.com

SCIENCE DIRECT®

Science of Computer Programming 53 (2004) 3–24

Science of Computer Programming

www.elsevier.com/locate/scico

An interactive environment for beginning Java programmers

Kenneth J. Goldman

Department of Computer Science and Engineering, Washington University in St. Louis, One Brookings Drive, St. Louis, MO 63130, USA

Received 25 August 2003; received in revised form 3 February 2004; accepted 17 February 2004

Available online 10 June 2004

Abstract

Building upon years of evolution in object-oriented programming language design, Java has emerged as the language of choice among many educators for teaching introductory computer science. A clean, type-safe language, Java provides a garbage collected heap and a comprehensive exception-handling mechanism. However, in spite of this support, many students in introductory computer science courses still find programming to be an overwhelming source of frustration. Linguistic concerns and programming mechanics demand so much attention that deeper concepts are often postponed for later courses, leaving students in introductory courses with the mistaken impression that computer science is a shallow discipline, concerned only with transcribing ideas into code, and not with the ideas themselves.

JPie is a tightly integrated programming environment for live software construction in Java. JPie treats programming as an application in its own right, providing a visual representation of class definitions and supporting direct manipulation of graphical representations of programming abstractions and constructs. Exploiting Java's reflection mechanism, JPie supports the notion of a *dynamic class* that can be modified while the program is running, thereby eliminating the edit–compile–test cycle. Following years of experience using Java as the vehicle for teaching introductory computer science, we have designed JPie to provide a more natural and fluid software development process that both raises the level of abstraction and eliminates many of the common pitfalls that beginning Java programmers face. This paper studies JPie from an educational perspective. We systematically review key programming abstractions and explain how JPie supports them in ways that keep beginning programmers focused on important ideas. Our experience using JPie in an introductory computer science survey course for non-majors is briefly discussed.

E-mail address: kjg@cse.wustl.edu (K.J. Goldman).

0167-6423/$ - see front matter © 2004 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2004.02.002

1. Introduction

College-level introductory computer science courses serve two competing objectives, education and training. On the education side, we want to expose students to the beauty of computer science, to provide insight into important ideas and ways of thinking, and to cultivate their intellectual curiosity about the field. We want beginning students to understand and appreciate powerful and inspiring ideas, in areas such as software architecture, algorithm and data structure design, and concurrency. On the training side, we want to instill programming and debugging skills in order to reinforce concepts and prepare students for later courses and beyond. To satisfy training objectives, however, introductory computer science courses are often reduced to language courses, in which students spend the bulk of their time learning linguistic constructs and programming mechanics. Low-level concerns undermine deeper educational objectives, leaving introductory computer science courses as "rites of passage" in which only those willing to contend with a steep learning curve in an artificial syntax survive. Such courses demand a bottom-up learning style and appeal to a small fraction of the college population, leaving the rest without an attractive opportunity to gain meaningful hands-on experience with computer science concepts.

1.1. Evolution of programming abstractions

High-level languages have steadily evolved to make software development more accessible. Languages like Fortran, C, Pascal, Smalltalk, Lisp, C++, and Java have succeeded because of their support for *abstractions* that make the programming process more natural and that can be mapped, by a compiler or interpreter, to efficient executable code. Procedural abstraction, parameters and return values, abstract data types, encapsulation, iteration, objects, methods, class hierarchies, inheritance, and polymorphism are examples of abstractions that have driven the evolution of high-level languages.

These abstractions transcend programming languages. They support programming models that allow people to think about computations at a higher level. Each high-level language provides constructs for expressing the abstractions in its programming model. The linguistic constructs are designed to provide compact and unambiguous expression of these abstractions in a way that is both readable by humans and efficiently processed by a compiler. However, it is the abstractions, not the linguistic mechanisms, that account for the progress in high-level language design, and it is these abstractions that form the most critical educational component of introductory computer science courses.

1.2. Beginning programmers focus on language details

Unfortunately, the training that must occur before students can study anything "interesting" often eclipses the abstractions and ideas that computer science educators try to emphasize. In order to make use of powerful programming abstractions, students first must learn to *describe* them in "code" and must learn to work within an edit–compile–test regimen that provides delayed feedback. Getting used to the syntax, the tools, and the programming and debugging process can easily take students most of a semester. On the other hand, if programmers could directly manipulate high-level abstractions and see the results immediately, then less time could be devoted to mechanics and more to intellectual

pursuits, resulting in inspiring introductory courses that appeal to a much wider population of students.

1.3. Direct manipulation of live object-oriented programs

We have developed a tightly integrated development environment, called **JPie** (Java Programmer's Interactive Environment), that supports live interactive object-oriented software development in Java [10,14]. JPie provides a close coupling of (1) a graphical editor supporting direct manipulation [17] of a program's semantic units, and (2) an execution environment in which program modifications take effect immediately on the running program. JPie's interactivity rests on the notion of a *dynamic class* [13], whose interface and implementation can be modified live, affecting even existing instances of the class. Dynamic classes fully interoperate with compiled classes. Consequently, JPie users have available the entire Java API (version 1.4), may create dynamic classes extending either dynamic or compiled classes, and can override methods on the fly. Instances of compiled classes may hold type-safe references to instances of dynamic classes, and may call methods on them polymorphically.

JPie provides graphical representations of programming language abstractions that expose the Java execution model and make software development immediate and tangible. JPie programmers directly manipulate the graphical representations to effect changes in the running program. Many operations, such as variable and method declaration and use, are accomplished by drag-and-drop. Through dynamic classes, JPie has fine-grain awareness of program structure and takes advantage of this knowledge to constrain program editing, check and maintain consistency, provide timely feedback, and eliminate the edit–compile–execute cycle. JPie's integrated debugger, which uses the same graphical representation, allows logical errors (including exceptions) to be handled on the fly.

Some argue that it is "good discipline" for students to learn how to express themselves clearly in textual code. However, we do not insist that children learn to write when they are first learning to speak. Similarly, the educational philosophy of JPie is to build up the intellectual foundation first, and introduce textual programming later. More specifically, we provide students with an interactive environment that supports a direct and immediate way of working with programming abstractions. Within that environment, students build an understanding of the programming model through the experience of creating software using those abstractions. Then, when it is time to learn how to express the ideas textually, students will already have an understanding of what they are trying to accomplish. Furthermore, if the interactive environment transparently exposes the execution model of the underlying language and provides tools for seeing the relationship between the visual representation and the corresponding textual implementation, then the transition will be even smoother. Textual programming could be introduced part way through an introductory course, once the basic foundations are understood, or dovetailed throughout the semester as each concept is mastered.

1.4. Why Java?

We chose Java as the basis of JPie, as well as for our introductory courses, because it provides clean type-safe support for standard object-oriented programming techniques

and abstractions and simplifies programming by providing a garbage collected heap and a comprehensive exception-handling mechanism. Furthermore, Java's extensive reflection mechanism, which provides access to detailed type information at run-time, enabled our implementation of dynamic classes and, consequently, live software development.

JPie was designed to support a more fluid software development process, not only to improve programmer productivity, but also to enhance the quality of computer science education. Having taught introductory computer science courses with Java since 1997, we had considerable experience working with students to help them write and debug Java programs. Therefore, we approached the JPie design with first-hand knowledge of the pitfalls that beginning programmers face, and we set out to build an interactive programming environment that would alleviate the problems associated with a compiled textual language so that more time could be devoted to teaching the conceptual foundations of computer science. At the same time, we wanted to expose students to the programming model of a widely used object-oriented language, provide access to the API of that language, and support a smooth transition to textual programming.

The remainder of the paper is organized as follows. Following a discussion of related work (in Section 2), we take a systematic look at the abstractions and mechanisms provided by object-oriented languages, paying particular attention to common pitfalls beginning programmers face. We explain how JPie supports these abstractions while alleviating the pitfalls. The discussion is divided into programming mechanics (Section 3), fundamental abstractions (Section 4), and object-oriented concepts (Section 5). Section 6 briefly describes our classroom experience with JPie. We conclude, in Section 7, with a summary and plans for future work.

2. Related work

In this section, we briefly explain JPie's relationship to related work in visual languages and integrated development environments.

2.1. Visual languages

Visual languages are designed for programming environments in which software is constructed by direct manipulation of graphical language primitives and operators. They are typically *new* languages (as opposed to graphical front-ends for existing languages), and they are generally based on execution models that are deemed to be particularly well suited for visual expression. Usually, these languages provide a tight integration of editing and program execution, and in some cases the program can be edited "live", while it is running.

Visual languages have been constructed using a variety of paradigms. One of the most common paradigms is dataflow, in which data flows across "arrows" to trigger actions performed in "boxes". Some examples of dataflow visual languages are Show and Tell [18], one of the first dataflow languages designed to be accessible to children; Prograph [7], which has been developed commercially; Khoros [21], which has been targeted for image and signal processing; and our own distributed application configuration language [19]. Dataflow is not the only visual language paradigm. Forms/3 [5] introduces procedural

abstraction within a declarative programming spreadsheet paradigm. ThingLab [4] uses a constraint-oriented paradigm to support the construction of geometric models. Statecharts [15] supports software development with a nested state-machine paradigm. VIPR [6] uses arrows and nested rings to declare program behavior with elements of object-oriented programming. AgentSheets [22,23] provides a rule-based paradigm for specifying how interacting agents gather and process information. Stagecast (a commercial realization of KidSim [8] and Cocoa [16]) uses a combination of rule-based programming and programming-by-example to support children in developing video games and simulations. Logo [20], another well-known programming language for children, uses a mixture of visual components and textual programming.

Colleges and universities generally have not adopted visual languages. This is not only because most professional programmers prefer textual languages over visual languages for general-purpose programming, but more importantly because the thought process involved in constructing programs within visual languages is often so strikingly different from that used in writing textual programs that knowledge transfer would be limited. It would be hard to justify a curriculum asking students to invest time and effort to learn a visual programming model that would be abandoned in the next course.

In designing JPie, we took a different approach to interactive software construction. Rather than design a new visual programming language, we chose to treat the programming process itself as an application domain, just as other graphical WYSIWYG applications support various other application domains. JPie's visual representation is not a new language, but instead serves as a language front-end for Java. As such, JPie benefits from years of accumulated research and experience in programming language design. JPie programmers learn to work within a standard object-oriented programming model and leverage the entire Java API, enabling a smooth transition into textual programming and making live software construction in JPie a viable alternative to textual programming in introductory courses.

2.2. Integrated development environments (IDEs)

Integrated development environments (IDEs) are general-purpose programming tools designed to improve the productivity of professional programmers. Essentially umbrella applications, they combine project file management, an editor, compiler, run-time system, and debugger under one roof. Some support multiple programming languages. IDEs that support software development in Java include Inprise JBuilder, Sun ONE Studio, NetBeans, Eclipse, Webgain Visual Café, Metrowerks Code Warrior, and many others. Some sophisticated IDEs, such as Eclipse, allow functionality to be added to the IDE as third-party plug-ins [9]. Simpler IDEs specifically targeted for the classroom include BlueJ [2] and DrJava [1], which also provides some features as an Eclipse plug-in. These IDEs support experimentation with Java through manual invocation of methods on objects and interactive evaluation of Java expressions. All use text editing as the primary means of software development.

IDEs support writing textual code in a number of ways. For example, source code editors may provide syntax colorizing, delimiter matching, and method name completion. When compile or run-time errors occur, the environment highlights the line of text at

which the error was detected. Common project management tasks are automated, and programmers do not have to separately invoke an editor, compiler, and debugger. However, they do not use a fine-grain internal representation of classes to maintain global syntactic consistency throughout the coding process. Although some provide interactive expression evaluation, IDEs do not eliminate the edit–compile–execute-cycle for general program development.

In addition to the basic tools, environments usually include a graphical user interface (GUI) builder for rapid layout of graphical components. Construction of the GUI is often "live" so changes in relationships between the GUI and the underlying data model are reflected instantly. However, in the end, a programmer must write the essential functionality of the application in a textual language. Moreover, although they are helpful for experienced programmers, GUI builders in IDEs can overwhelm inexperienced programmers by requiring them to complete the functionality of their applications by modifying computer-generated source code that beginning programmers are likely to find confusing.

JPie differs from a traditional IDE because it is one tightly integrated program, rather than a collection of loosely connected components. It provides a visual representation of the entire application, including not only the GUI, but the class definitions as well. The programmer never needs to drop down into textual programming. All aspects of the program can be modified directly while the program is running, and the system maintains a fine-grain internal representation of the program in order to prevent syntax errors and maintain consistency across the entire application. At the same time, JPie's visual mechanism provides access to compiled Java classes (including the Java API and third-party classes), and supports full interoperation between compiled and dynamically modifiable classes.

3. Mechanics

The abstractions and ideas that introductory computer science courses emphasize are easily eclipsed by linguistic details and programming mechanics in the minds of beginning programmers. A multitude of questions, often about how to express simple ideas, clouds the thought process and obscures the "big picture". In this section, we explain how JPie provides a more fluid software development process that alleviates many of the issues of programming mechanics that beginning programmers face.

3.1. Program editing

When writing and editing Java text, some of the questions students ask are purely syntactic ("Do I need a semicolon here?"). More problematic questions involve seemingly minor differences in syntax that imply vastly different semantics and may not be caught by a compiler ("Do I restate the type when I use an instance variable inside a method?" and "Should I use 'x' or 'this.x' to access the variable?").

We claim that these sorts of questions arise from an under-constrained editing process that forces programmers to *describe* abstractions in "code", rather than manipulate them directly. The problems are compounded by delayed feedback that results from the

edit–compile–execute cycle. Inexperienced programmers are fearful of describing things incorrectly ("What should I type?"). Inevitably, when they do describe things incorrectly, they become frustrated ("That's not what I meant!"). To make matters worse, delayed feedback can result in a student investing significant time and effort, perhaps making similar errors repeatedly, before discovering the problem. A student in an introductory programming course may become so bogged down in details to miss the important ideas and reach the mistaken conclusion that computer science is nothing more than arcane syntax.

3.1.1. Syntax errors

JPie provides a direct and immediate way of working with program abstractions, so it is impossible to create syntax errors. Variables and methods are declared by drag-and-drop, and expressions are built up using drag-and-drop or selection from lists. Parameter lists are kept consistent between definition and use. JPie keeps an expected type for each expression (e.g., actual parameter slots expect an expression that is assignable to the formal parameter type), and provides immediate, but unobtrusive, feedback if the current expression does not match the expected type.

3.1.2. Order of operations

Beginning programmers are sometimes confused by implicit rules about the order of operations. For example, students find it surprising at first when the Java expression

```
"The sum is"  + 3 + 5
```

has the result

```
The sum is 35
```

instead of 8, as expected. Programmers eventually grow accustomed to precedence rules, but with so many more important things for beginning programmers to think about, we decided to make execution order explicit in JPie. JPie does not use textual delimiters. Instead, nesting and execution order are represented explicitly as nested boxed. Execution order is further reinforced within the JPie debugger, where each expression is highlighted upon execution.

3.1.3. Assignment

Assignment statements in procedural languages pose unique concerns for beginning programmers. Those accustomed to mathematics think of the equal sign as a statement of equality, in which it does not matter whether a variable is on the left or right side of the expression. Furthermore, they see a statement of equality as a statement of truth that continues to hold throughout the program, and resist the idea of an equal sign indicating a possibly temporary assignment to a variable. ("I made '$x = 3$' up there, so why is it something different now?") Further complicating the situation are the fact that assignment statements are an exception to the usual left to right execution model, and confusion over the use of "=" in Boolean tests where "==" belongs.

To circumvent all of these problems, JPie represents assignment using a left-to-right statement in which an arrow, rather than an equal sign, indicates that the value of the expression on the left is being placed in the variable on the right, as shown in Fig. 1.

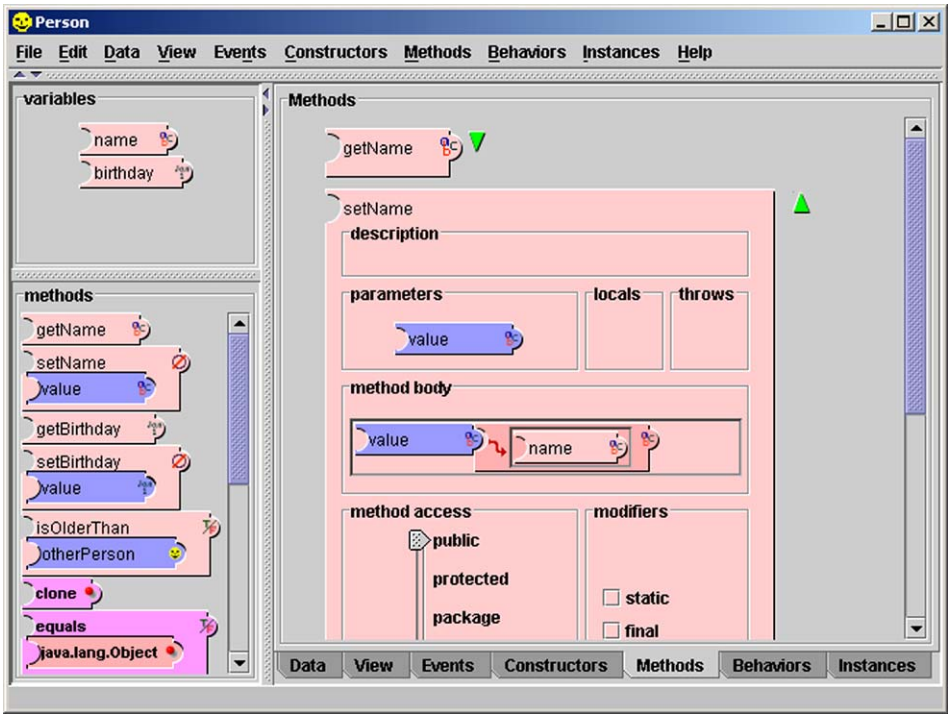

Fig. 1. The variable and method capsules of this Person class are declared and used by drag-and-drop. The variables and methods summary lists at the left include both declared and inherited members, which are distinguished by color. Variable scope is indicated by color as well. All modifiers are represented explicitly.

This solution does depart from the traditional notation, but we believe that it is a syntactic difference that can be easily overcome once a student is used to the programming model and is ready to transition to textual programming.

3.1.4. Modifiers

As students progress through their first course using Java, they learn about the meaning of various modifiers (such as static, final, synchronized, public, private, and protected). However, they often forget their meanings or forget to use them when appropriate. Rather than rely on memory, JPie provides a modifier panel as part of each instance variable and method of a class. Access modifiers are shown along a continuum and other modifiers are shown as checkboxes, as in Fig. 1. Moving the slider or placing the cursor over a checkbox provides text to remind the programmer exactly what that modifier means. This is in keeping with the overall philosophy that the programming environment should support the thought process by making everything as explicit as possible.

3.2. Execution, testing and debugging

One of the most compelling reasons for adopting JPie in the classroom is that software development is live. Beginners, perhaps more than seasoned programmers, make many

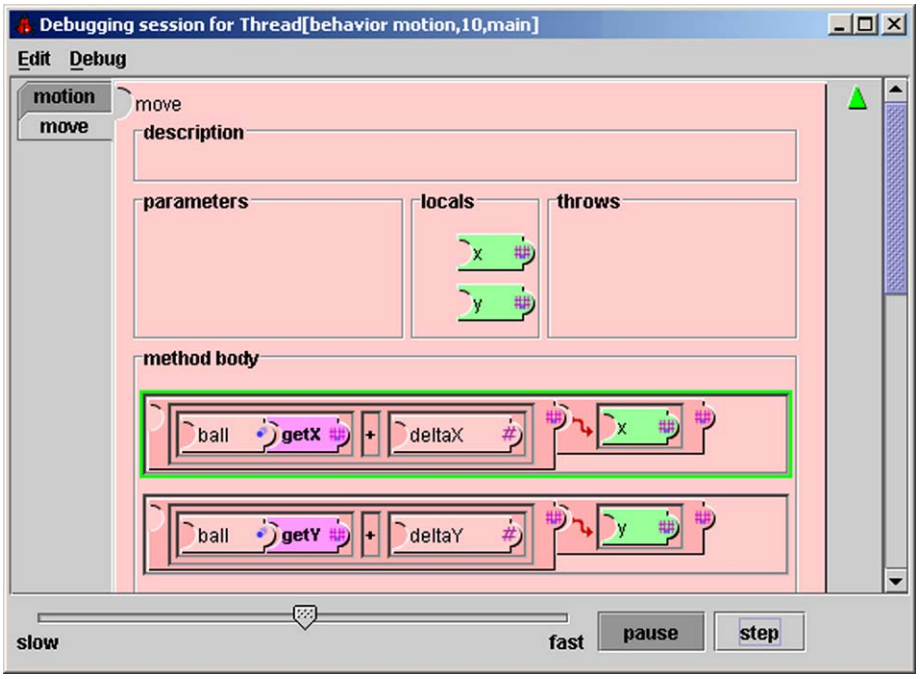

Fig. 2. The JPie debugger supports observation and modification during program execution.

small changes to programs and see how they manifest themselves in the running application after each change is made. In a traditional programming environment, each such change requires a recompilation step, restarting the application, and then driving the execution to the point at which the effect of the change can be observed. In JPie however, the edit–compile–test cycle is eliminated. Instead, programmers directly manipulate programs while they are running and can see the effects immediately, without restarting the application. This rapid feedback not only saves considerable time, but also creates an environment in which beginners are encouraged to learn through experimentation.

3.2.1. Error localization

Beginning programmers can literally spend hours locating logical errors that an experienced programmer would spot almost immediately. Partly because standard debugging tools can be confusing or cumbersome, programmers often resort to inserting print statements into their programs in order to determine what is happening during the execution. The JPie debugger (Fig. 2) assists in error localization by allowing programmers to watch the execution unfold in "slow motion", to place the cursor over expressions to see their values, and to pause the execution at any time to look at the effects on object instances. Also, because it is thread-oriented, programmers can concentrate on errors in one part of a program while the rest of the application continues to run. When an error is located, programmers can modify the program in the debugger using the same familiar graphical representation, and continue execution with the modifications in place.

3.2.2. Exception handling

When an exception occurs in a typical Java program, the exception is propagated out to a corresponding catch clause, or all the way up the stack if there is no matching catch clause. When this happens, the programmer can read the stack trace to reconstruct what happened, but there is little hope of resuming execution. The philosophy of JPie, however, is to try to give the programmer every opportunity to correct errors in the running program. To this end, JPie provides an option whereby the debugger comes up immediately when an exception is first thrown, giving the programmer the option to either (1) continue—propagate the exception, perhaps declaring it as being thrown from the method, (2) handle the exception—insert a catch clause to handle the exception, or (3) try again—execute the offending statement again after modifying the program, presumably so that the exception will not occur again. This level of support for exception handling not only saves significant time in fixing program errors, but also encourages a more fluid development process in which one can begin by programming for the expected case and add the exception handling code later. To support this form of incremental development, JPie does not insist that calling methods handle, or declare to be thrown, all exceptions that could result from each method call.

3.3. Project management

Typical IDE's provide a project management facility that allows the programmer to include certain classes in a project, identify a "main" class, and specify the classpath and various other properties of that project. For introductory courses, we have found that such mechanisms simply add another level of complication onto an already overburdened development process. Therefore, in JPie, we simply encourage programmers to use Java's existing package mechanism to organize their work. Also, since any class with a no-argument constructor can be manually instantiated in JPie, there is no notion of a "main" class until the programmer is ready to export the completed application to be run outside the JPie environment.

One benefit of a project management facility for introductory courses is the ability to provide students with a partially completed project. To support this mechanism in JPie, we allow a remote directory or URL to be specified as a project source. When JPie starts, it examines that source and compares it with the local working directory and downloads any missing files and folders so that they will be available for the student to modify. We have found this method of project distribution vastly simplifies course management.

3.4. Transition to textual programming

Graphical representations, such as those provided by JPie, are not as compact as textual languages. As programmers become more advanced, it makes sense to transition from a graphical representation to a textual one. To support this, JPie provides a menu option to show the source code for a dynamic class so that the JPie programmer can see the relationship between the graphical representation of what they have specified and its corresponding textual implementation. In providing this feature, we took great care not to simply churn out code that runs, but to generate code in style we would find acceptable for a student to write in a traditional introductory course. We felt that giving

JPie programmers "clean" example code not only would help ease the transition to textual programming, but also would help instill the notion that textual source code should be clear and understandable.

4. Fundamental abstractions

Having discussed programming mechanics, we now discuss fundamental programming abstractions, paying particular attention to common pitfalls facing beginning programmers and how JPie addresses these problems. These fundamental abstractions include naming, data abstraction and encapsulation, procedural abstraction, and control flow. Section 5 provides a similar treatment of abstractions unique to object-oriented languages.

4.1. Naming abstraction

Beginning programmers learn that names serve both as identifiers and as documentation. As simple as naming may seem at first, programming errors often arise due to confusion over the type of an identifier, inconsistent renaming, name clashes, identifier masking, and the distinction between declaration and use. In the following sections, we describe how JPie uses a combination of explicit representation and direct manipulation to address these problems.

4.1.1. Awareness of type and scope

Textual identifiers by themselves do not directly convey information about scope and type. Forgetting the type of an identifier results in compile errors and often requires rethinking the offending expressions. Worse, mistakenly using the variable from the wrong scope (due to masking) results in logical errors that are often hard to find. In JPie, each identifier explicitly presents not only its name, but also its type and scope. This saves programmers from relying on their memories for this information or having to look back carefully through the program text in order to find the closest declaration. The identifier for each variable, method, and constructor is represented visually in JPie as a capsule. The name is shown as a label on the capsule, the scope is indicated by the capsule's color, and the (return) type is represented by an icon at the right edge of the capsule, on a protrusion that we call the "dot". (In addition, the fully qualified type name appears as pop-up text when the cursor is placed over the icon.) The capsule shape is designed so that capsules can be linked together in a chain to form type-safe expressions, much as textual identifiers are chained together using dot notation in the textual representation, but with the dot additionally providing type information.

4.1.2. Definition and use

The difference between defining and using a program entity is an important concept that beginning programmers must learn. This distinction pervades every aspect of programming, including variable declaration versus variable access, method definition versus method call, and formal parameter declaration versus passing actual parameters. Confusing declaration and use is an extremely common source of program errors. For example, we have often observed students trying to assign to instance variables within

a method or constructor, but instead mistakenly restating the type of the variable and simply initializing a local variable on the stack, leaving the instance variable unchanged. For example,

```
class Box {
    int width = 0;
    int height = 0;
    setSize(int w, int h) {
        int width = w; // unintended declarations
        int height = h; // can be hard to track down
    }
}
```

Errors like these have less to do with the thought process of the programmer than with experience in the syntax of the language. Clearly, the programmer did not intend to declare local variables, but was instead confused by the syntactic similarity between declaration and use. Most likely, the programmer either did not realize that restating the type of the variable resulted in a new declaration, or simply restated the type of the variable out of habit. Nonetheless, these kinds of errors occur frequently and can cause hours of frustration for a beginning programmer.

JPie supports both declaration and use through drag-and-drop manipulation of program elements. Declaration is accomplished by dragging a type from its "Packages and Classes" window into a particular location or scope in the class definition window. For example, dragging the String type into the "Data" panel of a class declares an instance variable of type String. (Similarly, dragging the String type into the "Methods" panel declares a method with String as the return type.)

Once a field, method, parameter, or local variable has been declared, it can be dragged into an expression for use. Because the choice is made by direct manipulation, there can be no confusion over which variable is meant while the expression is being constructed, so masking is not an issue. To assist in this process, JPie maintains a visual display of the variables accessible from the currently selected expression, and allows a variable to be dropped into an expression only the expression is within the variable's scope. Once the expression has been formed, the chosen variable's scope remains evident in the color of the capsule, preventing any confusion due to name masking.

Note that the programming model is identical to that in the textual language. Declaration is accomplished by specifying the type, and use is accomplished by placement of the identifier. The difference is that both definition and use are direct and tangible in JPie. Accidental redeclaration is extremely unlikely because the act of using a variable or method (simply dragging it into the expression) looks and feels so different from declaring one (dragging the type into a region and seeing the new variable or method appear).

4.1.3. Renaming

Names are critically important for documentation and program understanding. However, the risk of program errors resulting from inconsistent renaming may dissuade beginning programmers from improving their existing identifier names. JPie avoids this pitfall because variables and methods are directly manipulated. The programming

environment is aware of each use of a variable or method, so it automatically renames all uses whenever the name is changed in the declaration. Similarly, JPie keeps the names of accessor and mutator methods consistent with the names of the instance variables, as discussed in Section 4.2.4.

4.2. Data abstraction

Introductory courses emphasize the importance of creating an abstraction barrier around the data inside of objects so that the internal representation is not exposed and can be changed over time without affecting other parts of the application. Students often try to "save time" by declaring instance variables public and accessing them directly from other classes. Other mistakes involve forgotten or incorrect initialization of encapsulated data so that representation invariants are violated and other methods fail. These can be hard for students to track down because the errors occur in other methods, not in the places where initialization was needed.

4.2.1. Encapsulation

The act of instance variable declaration in JPie emphasizes the concept of encapsulation because the programmer drags the desired type into the class where a variable is created. Seeing the variable created within the class window helps programmers appreciate that the data in that variable is actually held within objects of that class. Textual representations clearly show instance variables within the outermost braces of the class declaration, but they do not create quite as powerful a visual impression of encapsulation.

4.2.2. Initialization

Beginning programmers often forget to initialize variables. Each instance variable declared in JPie has an initialization expression that the programmer can edit. When first declared, the initialization expression explicitly shows the default value that would be assigned by Java for the corresponding type. Because the initialization expression is explicit, mistakes resulting from forgotten initialization are less likely.

4.2.3. Constructors

A surprisingly common mistake among beginning programmers is to pass parameters into a constructor and then forget to assign their values to the corresponding instance variables. Because this form of initialization is such a common idiom, JPie allows instance variables to be marked "supplied" to indicate that their values are supplied to the constructor. While an instance variable is marked "supplied", each new constructor created will automatically have a parameter that is assigned to that instance variable within the constructor.

When extending a class, students often forget to call a parent constructor. To prevent this mistake, each constructor in JPie begins with a call to a constructor of the parent class for proper initialization of inherited fields. The programmer may change which parent constructor is called (or call another constructor within this class), but may not delete the call entirely.

4.2.4. Accessors and mutators

Instance variables in JPie are declared private by default, and public accessor and mutators (get and set methods, in accordance with the JavaBeans convention) are created by default. Although the programmer may change the access modifiers, and even delete the accessor or mutator, the fact that these are created automatically makes it easier for the programmer to get in the habit of proper encapsulation. Like any other methods, the bodies of the accessor and mutator may be modified. However, their parameter lists may not be changed and their names are automatically kept consistent with the name of the corresponding instance variable. Default accessor and mutator methods for a String variable are shown in Fig. 1.

4.3. Procedural abstraction

Having been exposed to mathematical functions, most students readily accept the idea of a procedure that takes one or more arguments and return a result. However, they are not accustomed to the idea of strong type checking, nor are they familiar with the idea of a method running within an object and having access to its members. Furthermore, they often exhibit confusion between the method definition and the method call, sometimes trying to declare the types of parameters within the method call itself. We discuss these in turn.

4.3.1. Parameters and return values

The most common mistakes in method calls concern the order and types of the parameters, as well as type errors resulting from incorrect use of the return values. Also, within the method definition, beginners will sometimes forget to return a value of the right type. To address these problems, each method call in JPie is represented as a capsule with labeled slots for its actual parameter expressions. Whenever the parameter list in the method signature is changed or reordered, the parameter lists in the method calls are updated accordingly so that consistency is always maintained. Each parameter slot has an expected type, which is explicitly represented as an icon, and type checking occurs as expressions are being constructed. Each method whose return type is not void has an explicitly designated return expression, so there is no danger of forgetting to return a value.

4.3.2. The implicit parameter "this"

For students accustomed to mathematical functions, it is sometimes hard to internalize the idea that a method is part of a class and has access to the data members of the object on which it was called. For example, when creating a transfer method on a bank account class, students often ask why only one account (the destination) must be passed as a parameter, and we must remind them that the transfer will be called on a bank account instance, which is the source of the transfer.

There are several ways in which JPie reinforces the idea that methods in Java are defined within the context of a class, and that execution takes place within the context of an instance of that class. First, as described in Section 4.1.1, JPie continually maintains a list of variables accessible within the scope of the selected expression. So, when editing the method body of the bank account's transfer method, students see a list containing not only the parameters and local variables of the method, but also the balance and any other instance variables defined in the bank account class. Second, each new expression in JPie

provides an icon representing the type of the class itself. From that icon, one can pull down a menu in order to access the members of the class. Both of these mechanisms reinforce the idea that methods do not stand on their own, but are part of a larger class definition and, as such, have access to other parts of that class. Finally, JPie's "Instances" panel provides a list of instances of objects that have been created. On that panel, instances may be select for viewing, and arbitrary expressions can be executed within them. This makes it clear that execution takes place within a particular object.

4.3.3. Method declarations versus method calls

As discussed in Section 4.1.2, the distinction between definition and use is made concrete in JPie, so there is no danger of confusing the two. Calling a method is accomplished either by dragging the method into an expression or selecting it from a pop-up list. Within the method call itself, labeled and typed slots are provided for the actual parameter expressions, and JPie keeps these consistent with the parameter list defined within the method.

4.4. Control flow

Beginning programmers must learn to relate a static representation to its realization as an executing process. As the control flow becomes increasingly complex (with iteration, recursion, threads, and event handling), it becomes increasingly difficult for students to imagine the possible executions of the program, diagnose errors, and predict implications of program modifications. This section describes aspects of JPie that are designed to make control flow more obvious and to help programmers with this thought process.

4.4.1. Representation

JPie provides traditional control flow constructs, such as "if" and "while", a generalized "switch" statement (called "match") in which the cases can be arbitrary expressions, and a "for each" statement that supports iteration over Java collections and ranges of values. To make the representation more tangible, JPie replaces the traditional textual delimiters (curly braces) by nested boxes, as shown in Fig. 3. This elucidates, for example, what statements are governed by a conditional test or comprise a loop body.

4.4.2. Observing execution

To assist students in making the leap from the static representation to the active process, the JPie debugger uses the same visual representation and allows the user to step through the execution expression-by-expression, not line-by-line as is typical of most debuggers. This allows fine-grain observation of the execution to help students understand the order in which expressions are evaluated. For example, one can observe the arguments of a method call being evaluated before the method itself is called. The expression that is about to execute (or is currently executing) is highlighted in green, and the most recent value of each expression shows as pop-up text when the cursor is placed over the border of that expression.

The JPie debugger shows all stack frames in a tabbed panel, so that it is easy to see how the execution reached its current point. Moreover, since all currently executing expressions are highlighted in green, one can look back at the previous stack frame to pinpoint exactly which expression caused the current method invocation. This is particularly useful for

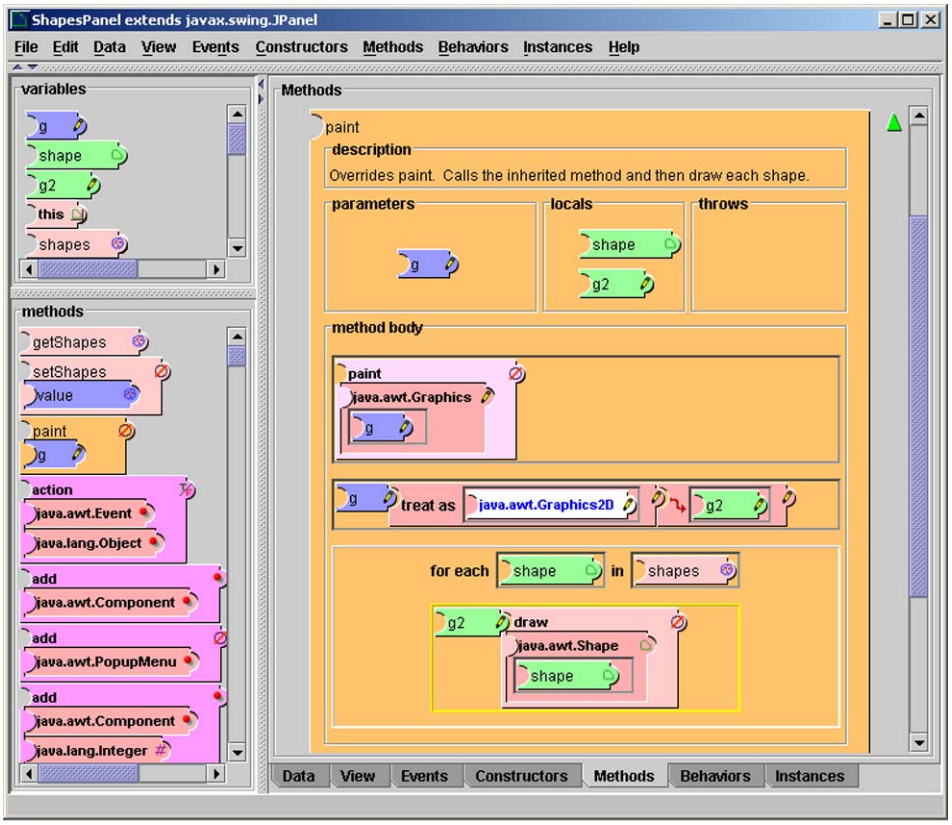

Fig. 3. Nested boxes, instead of delimiters, indicate scope.

understanding recursive methods, especially when used in conjunction with the pop-up text to see the values of the same expression in different stack frames.

4.4.3. Non-terminating execution

Beginners (and even experienced programmers) can waste considerable time finding and fixing non-terminating programs. To help programmers quickly identify infinite loops and unbounded recursion, JPie supports loop and stack bounding that will notify the programmer when the execution exceeds the bound, offering the choice to either abort or increase the bound and continue, perhaps after modifying the termination condition of the loop or the base case of the recursion in the debugger.

4.4.4. Threads, synchronization and deadlock detection

Although important in modern software development, creating and managing threads can be an issue for beginning programmers. A particularly elusive problem, even for more advanced programmers, is providing proper synchronization while avoiding deadlock.

JPie programmers may create threads in the usual way (by instantiating a Thread object and calling its start method). A simpler alternative is to construct a "behavior" that describes a periodic task to be carried out in its own thread. A behavior can be started manually in JPie, programmatically from JPie statements, or can be marked to "autostart" for each instance of the class. In general, when an error occurs in a thread (or a breakpoint is reached), the thread appears in its own window in the JPie debugger, while other threads continue to execute.

Any method in JPie may be marked as "synchronized", and the environment automatically provides deadlock detection. When JPie detects a deadlock, it displays a resource allocation graph showing the cycle, and gives the programmer an opportunity to open the offending threads in the debugger to see where they are blocked, in order to understand how to fix the problem. Also, threads may be selectively aborted in the debugger to break the cycle.

5. Object-oriented concepts

Having looked at the fundamental abstractions, we now describe JPie's support for object-oriented design and implementation, paying special attention to how JPie makes these abstractions more accessible to beginning programmers.

5.1. Classes and instances

The distinction between the class definition and the objects of the class is made obvious in JPie. As mentioned in Section 4.3.2, each class definition window has an "Instances" panel that maintains a list of instances of that class. On that panel, one can manually create a new instance, and select and view an instance, as well as execute arbitrary expressions within that instance.

5.2. Class hierarchies

5.2.1. Inheritance

In JPie, one can extend either a dynamic or compiled class simply by selecting it and then choosing the 'extend' option from the menu. Whenever a class window is opened, one sees a list of not only the declared fields and methods of the class, but also all of the inherited fields and methods accessible to the class (in a different color from the declared fields and methods). This makes inheritance a concrete, visible part of the programming process, rather than an implicit idea that must be remembered by the programmer. One can access and call inherited fields and methods by drag-and-drop, just as for declared members.

5.2.2. Method overriding

When writing Java text, one overrides a method by restating the signature of an inherited method. So long as the name, return type, and parameter types are consistent, and the access modifiers are permissible, the Java compiler will understand that the inherited method is to be overridden. If the return type or access modifiers are wrong, the compiler generates an error message. However, if the name is slightly misspelled or the parameter

list is not quite consistent, the compiler will simply not override the method. This can be a source of frustration for beginning programmers ("Why isn't my method being called?") who are not accustomed to checking every character. Furthermore, the signature matching approach to overriding fails when the parent method is renamed, and sometimes results in accidental overriding when the programmer unwittingly creates a new method with the same name and parameter list as an inherited method.

JPie avoids all of these issues by making method overriding a concrete act. To override a method in JPie, one drags the inherited method onto the "Methods" panel of the class. The new method is created with the same name and parameter list as the parent method, but it is shown in a different color to indicate that overriding has occurred. Furthermore, the name is henceforth kept consistent with that of the inherited method and the programmer is not permitted to change the name or the parameter list of the overriding method. Any changes to the name or parameter list of the overridden method are automatically propagated. Finally, the access modifier for the method, as well as its list of thrown exceptions, is kept consistent with that of the inherited method.

5.2.3. Polymorphism

When a method is overridden, it can be called polymorphically through variables of an ancestor type. Students often need to see polymorphism several times in various contexts in order to appreciate its power in object-oriented design. Through live modification, JPie helps make polymorphism concrete. When one overrides a method in JPie, the change takes place immediately, even on existing instances of the class in a running program. Therefore, students can experiment with overriding methods in a class and seeing how the instances of that class begin behaving differently as their methods are called polymorphically. Furthermore, when an overriding method is deleted, instances of the class immediately revert to the inherited method, so students can delete (and undo the delete of) an overriding method to see the difference first hand.

5.3. Model/view separation

Separating the data model from its user representation is standard object-oriented programming practice. Among other things, it allows multiple views of the same data to be open simultaneously and consistency to be maintained among them when the data model is changed. Java's swing package is based on model/view separation and is supported directly by JPie. Within JPie, one can use drag-and-drop to place graphics components into the "View" panel to create a view for the class. Connections can be formed, again using drag and drop, to relate the properties of the data model to those of the graphics components in the view, as shown in Fig. 4. Property change listeners are added automatically. When an instance is selected on the "Instances" panel, the view is shown for the selected object. When a view definition is changed (e.g., by adding or removing a component or a connection), all open views are updated accordingly.

5.4. Event handling

Java supports user interaction with a relatively sophisticated event source/listener model. Students must learn that components generate events, that listeners must be

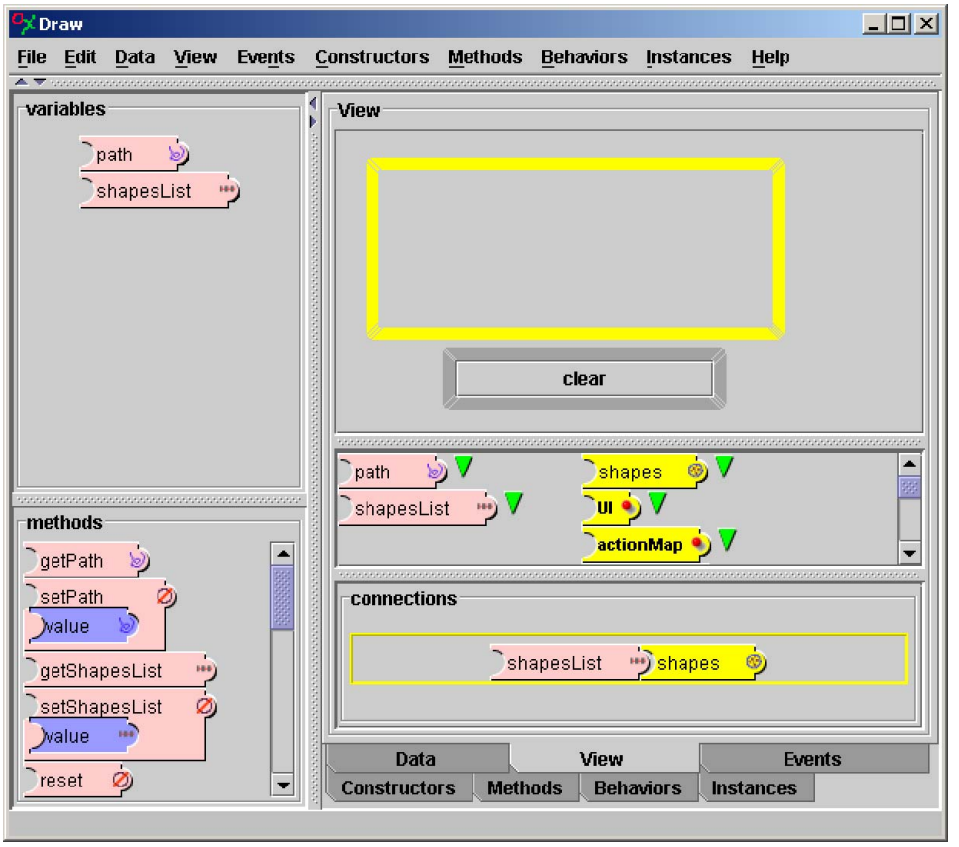

Fig. 4. Property connections relate the data model and the view.

registered to be notified of the events, and that the listener objects themselves must implement methods in particular interfaces order to respond to specific events. A thorough understanding of this framework is generally beyond the reach of a beginning programmer; yet building interactive programs that handle user input is an important motivator for beginning students.

JPie supports creating listeners and registering them with event sources in the usual way, but it also helps beginning programmers by offering a streamlined event handler construction process that simplifies, yet conforms to, the Java event model. The process begins by selecting the graphics component in the view that will be the source of the desired event, and then creating an event handler from an “Event” pull-down menu. The JPie programmer then presses a “record” button that begins recording all events that are generated by that component. After demonstrating the user event (such as a mouse click), the JPie programmer can select it from a list of the recorded events. Once the desired event is selected, JPie renames the event handler method accordingly (e.g., *mouseClicked*) and creates the appropriate formal parameter (e.g., *MouseEvent*). A conforming listener object

is automatically created and immediately registered with that component in the views of every instance. As with any other JPie method, the programmer can edit the body of the event handler, with changes taking effect immediately in the running program.

6. Classroom experience

We have been using JPie in the classroom since January 2003. The course, Washington University CS123, "Introduction to Software Concepts", has been designed from the ground up using JPie to expose students to important computer science concepts in a laboratory setting. All programming projects are completed during class time, with students working individually and in pairs. Because JPie allows direct manipulation of programming language abstractions, there is much less overhead in introducing important ideas. With no syntax to teach, students begin defining classes and methods from day one. Consequently, we expose students to many more "big ideas" than we can in our traditional Computer Science I course using textual programming in Java. For example, near the end of the course, students create a multithreaded client/server Internet chat application. That project, which takes only about three hours for the students to complete, involves client and server sockets, object streams, client message handler objects, user interface construction, and user event handling. Student reaction to the course, as gauged both informally and by official course evaluation forms, has been overwhelmingly positive. Systematic evaluations of the course, based on controlled learning assessments and analysis of videotaped student conversations during laboratory sessions, are planned. Further discussion of the CS123 curriculum is provided elsewhere [12]. A syllabus and details about the CS123 programming projects are available on the course web site [11].

We believe that JPie offers an opportunity to attract a more diverse student body to computer science courses. Research in learning styles indicates that gender plays a role in people's approach to computer programming [24] and that a top-down "bird's eye" view appeals more to women than the traditional bottom-up computer science course [3]. Interest in CS123 (an enrollment of approximately 50% women over three semesters) seems to support this.

7. Conclusion

JPie is a tightly integrated development environment that supports live construction of Java programs by direct manipulation of graphical representations of class definitions. JPie lends itself to introductory computer science education as an interactive environment in which students can learn the Java programming model while avoiding many of the common pitfalls and distractions of textual programming. JPie is an ongoing research project [14]. Plans for JPie in the near future include a class hierarchy editor, "mixed-mode" editing that allows a combination of textual and graphical program representation, and special-purpose integrated support for relational database access and development of client/server distributed applications using standard technologies. Educators interested in using JPie in the classroom are encouraged to contact the author.

Acknowledgements

I thank the following students for their contributions to the JPie project: Joel Brandt, Ben Brinckerhoff, Vanessa Clark, Melanie Cowan, Matt Hampton, Adam Mitz, Jonathan Nye, Sajeeva Pallemulle, and Richard Souvenir. I also thank the students in CS123 for valuable feedback. I thank Dan Kimura for advice in the area of visual programming, and for insightful discussions about the difference between education and training. I thank Joel Brandt, Ben Brinckerhoff, Sally Goldman, Adam Mitz, Keith Sawyer, and Harri Thorvaldsson for their comments on earlier versions of this paper.

This work was supported in part by the National Science Foundation under CISE Educational Innovation Grant 0305954.

References

[1] E. Allen, R. Cartwright, B. Stoler, DrJava: A lightweight pedagogic environment for Java, in: 33rd SIGCSE Technical Symposium on Computer Science Education, February, 2002.
[2] D.J. Barnes, M. Kölling, Objects first with Java: A practical introduction using BlueJ, Prentice Hall, Pearson Education, 2003.
[3] L. Blum, Transforming the Culture of Computing at Carnegie Mellon, November, Computing Research News, 2001.
[4] A. Borning, The programming language aspects of ThingLab, a constraint-oriented simulation laboratory, ACM Transactions on Programming Languages and Systems 3 (1981) 355–387.
[5] M. Burnett, J. Atwood, R.W. Djang, H. Gottfried, J. Reichwein, S. Yang, Forms/3: A first-order visual language to explore the boundaries of the spreadsheet paradigm, Journal of Functional Programming 11 (2) (2001) 155–206.
[6] W. Citrin, M. Doherty, B. Zorn, Formal semantics of control in a completely visual programming language, in: Proc. of 1994 IEEE Symposium on Visual Languages, St. Louis, 1994, pp. 208–215.
[7] P.T. Cox, F.R. Giles, T. Pietrzykowski, Prograph: a step towards liberating programming from textual conditioning, in: IEEE Workshop on Visual Languages, 1989, pp. 150–156.
[8] A. Cypher, D.C. Smith, KidSim: End user programming of simulations, in: Proceedings of CHI, Denver, May 7–11, ACM, New York, 1995, pp. 27–34.
[9] E. Gamma, K. Beck, Contributing to Eclipse, Addison Wesley, Boston, 2004.
[10] K.J. Goldman, A demonstration of JPie: An environment for live software construction in Java, in: OOPSLA'03 Conference Companion, Anaheim, CA, USA, October 26–30, 2003.
[11] K.J. Goldman, Washington University CS123: introduction to software concepts, since January, 2003, http://www.cse.wustl.edu/~kjg/cs123.
[12] K.J. Goldman, A concepts-first curriculum for introductory computer science, in: 35th SIGCSE Technical Symposium on Computer Science Education, March, 2004.
[13] K.J. Goldman, Live Software Development with Dynamic Classes (in preparation).
[14] K.J. Goldman et al., JPie: Programming is Easy, 2003, http://jpie.cse.wustl.edu.
[15] D. Harel, Statecharts: A visual formalism for complex systems, Science of Computer Programming 8 (3) (1987) 231–274.
[16] N. Heger, A. Cypher, D.C. Smith, Cocoa at the visual programming challenge'97, Journal of Visual Languages and Computing 9 (2) (1998) 151–169.
[17] A. Kay, The early history of Smalltalk, in: T.J. Bergin, R.G. Gibson (Eds.), History of Programming Languages - II, ACM Press, Addison-Wesley Publishing Co., New York, 1996, pp. 511–578.
[18] T.D. Kimura, J.W. Choi, J.M. Mack, A visual language for keyboardless programming, Technical Report WUCS-86-6, Washington University in St. Louis, June, 1986.
[19] T.P. McCartney, K.J. Goldman, Visual specification of interprocess and intraprocess communication, in: Proceedings of the 10th International Symposium on Visual Languages, VL'94, St. Louis, MO, October, 1994, pp. 80–87.

[20] S. Papert, Mindstorms: Children, Computers, and Powerful Ideas, Basic Books, New York, 1980.
[21] J. Rasure, C.S. Williams, An integrated visual language and software development environment, Journal of Visual Languages and Computing 2 (1991) 217–246.
[22] A. Repenning, Creating user interfaces with agentsheets, in: 1991 Symposium on Applied Computing, Kansas City, MO, IEEE Computer Society Press, Los Alamitos, 1991, pp. 190–196.
[23] A. Repenning, Agentsheets: a tool for building domain-oriented dynamic, visual environments, University of Colorado at Boulder, Ph.D. Dissertation, Department of Computer Science, 1993.
[24] S. Turkle, The Second Self: Computers and the Human Spirit, Simon and Schuster, New York, 1984, pp. 108–119.